Considering the DØ EDM

Issues and Solutions in the DØ Event Data Model

Presented for the ATLAS Architecture Meeting

13 August 1999

Marc Paterno

Fermilab / CD Special Assignments

Preface

- I hope for more of a discussion session than a lecture.
- I've included some material of an introductory nature for any audience members whose C++ expertise is limited. If I dwell on anything too elementary, please tell me to move on.
- Look out for "buzzword alerts", where (if asked) I'll pause to describe buzzwords.
- Some slides contain hyperlinks, useful if you're viewing this on the web.

Outline

I shall present the general problems that the DØ EDM attempts to solve:

- ⌘Some problems are inherent to handling event data (*e.g.*, keeping of "bookkeeping" information).
- ⌘Some problems are features of our choice of programming language. DØ is an almost purely C++ shop.

I shall also say where I think the DØ EDM (and some related systems) can be improved.

See also the newly updated EDM Tutorial, at

http://cdspecialproj.fnal.gov/d0/EDMTutorial/welcome.htm

What Does the Event Do?

- Encapsulate all the data from one collision
  - ⌘Must be a heterogeneous collection
  - ⌘Provide uniform access to elements it contains
  - ⌘Provide type-safe access to the elements it contains
  - ⌘Control the lifetime of the elements it contains
- Provide the mechanism through which software modules communicate
  - ⌘Provide a way to minimize physical coupling between reconstruction algorithms
  - ⌘Allow saving the state of reconstruction at any step

What Else Does the Event Do?

Provide “bookkeeping” information for all reconstruction output

- Provide support for assignment of identification tags
- Prevent users from modifying results already labeled

Also, the Event must work smoothly with the persistence mechanism and with the reconstruction, triggering, and analysis frameworks.

Heterogeneous Collection

C++ is a strongly typed language.

The collections provided by the C++ STL can contain only objects of the same type.

Possible solutions:

- Contain each type explicitly
  - Every time we add a new type, we have to modify the Event class (the header file); every time we do this, we have to recompile all code that uses the Event (which includes the header). This is unmaintainable.
- Contain only pointers to a base class
  - Everything stored in the Event inherits from this class.

Buzzword Alert!

object: An object is an instance of a class. Typically, each variable refers to some object (or, in C++, it could be a basic type like int or bool or double*).

class: A "class" may be though of as a data type. The description of a class includes what types of data each instance will contain, and what functions are defined for manipulating those data.

Example: the class string. Each instance of string (each string object) contains a sequence of chars; two instances can have different sequences. Each instance has available the same functions, *e.g.* size() returns how many chars are in the string.

Buzzword Alert!

strongly typed: This means that each variable has a type, function arguments have types, and function return values have types. The C++ compiler uses strong typing to (try to) prevent you from “fitting the wrong plug into the socket”. I’d call Fortran and Java strongly typed; I’d call Smalltalk, LISP and Python untyped (because any variable can be given any value).

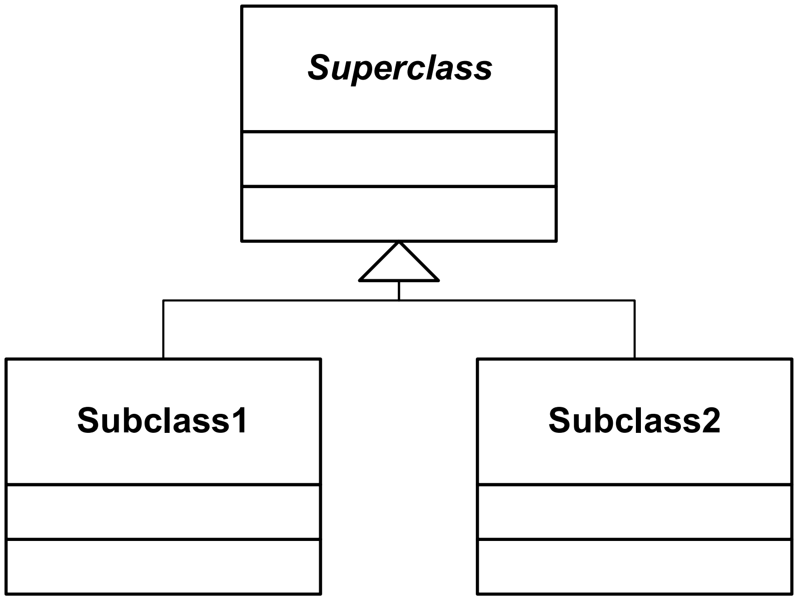
inheritance: One of the four basics of the “OO paradigm”. Here, we mean that each of the classes that inherit from the “base” class have a relationship that is known to the compiler, and they share some common functions.

Buzzword Alert!

subclass (also derived class): A class that inherits from another class, which is called the superclass.

superclass (also base class): A class from which another class inherits.

The diagram at right shows the standard (UML) notation for inheritance, and shows a superclass having two subclasses.



Back to the Mixed Collection

To get a heterogeneous collection in C++:

- Our "event data" classes (DØ calls them collectively chunks) inherit from one base class (AbsChunk). The prefix "Abs" indicates this is an abstract class.
- The Event contains chunks by reference, not by value (the Event holds a pointer to a chunk, not the chunk itself).

The Event doesn't need to know about all the classes it contains. The requirement of containment by reference will make issues of lifetime control more complex.

Buzzword Alert!

abstract class: A class of which one cannot make an instance. Abstract classes exist mostly to define interfaces.

interface: An interface class is an abstract class that contains only pure virtual functions (no data, no functions which are not pure virtual).

pure virtual function: A virtual function which is given no implementation; it must be implemented by a subclass.

virtual function: A function which the compiler knows can be overridden by the implementation in a subclass.

Uniform Access

- What we mean here is that we want the “look and feel” of access for each type of event data to be as similar as possible, so that the system is easier to comprehend.
- The only tradeoff here is that the access method must be of sufficient generality to meet all foreseeable needs.
- The DØ EDM provides the required mechanism, but this mechanism imposes some demands on the “chunks” (event data objects).

Type-safe Access

We don't want a user to get a bunch of muons when he requested a bunch of electrons.

- We'd like the compiler to prevent this, when possible.
- We'd like the run-time system to prevent this, if necessary.
- We never want such a mistake to quietly give an incorrect result.

This requirement is in tension against the concept of heterogeneous collections

- We want the Event to give us an instance of the right class, when it doesn't know what it contains.

Templates to the rescue

To resolve this tension, the DØ EDM makes use of templates

- TKey<T>, AbsTSelector<T>, and THandle<T>.
- This allows us to have the EDM classes provide the "ugly" code, and to have other classes perform simpler tasks.

The use of templates was critical to making the system type-safe: the class TKey<T> has to know about the type T it will get from the event, even if the Event doesn't know of it.

Buzzword Alert!

template: Templates are a mechanism by which we can write code once for a family of related functions or classes. Using templates, we can write a class template for the class list, which would provide the family of classes list<int>, list<AbsChunk*>, list<list<double> >, etc.

Drawbacks to templates

The use of templates has some drawbacks.

- Writing robust template code is somewhat trickier than writing “normal” code.
- Not all current compilers handle templated code well. Templates may add considerable complexity to your system for building software.
- Some popular HEP C++ products currently have trouble with templates.

Only the first of these is permanent. DØ decided the benefit was more important than the drawbacks.

Access to Chunks

Users “extract” items in the Event using keys, and are given read-only access through handles

-
  ```
  TKey<SVXChunk> svxkey(...); // arguments elided
  THandle<SVXChunk> h = svxkey.find(event);
  if (h.isValid()) h->svxChunkFunction();
  ```
- The key is responsible for knowing the data type it is to match (here, SVXChunk), and assuring that the returned object really is of this type.

Keys carry selectors, which are used to identify the specific instance(s) which the user wants.

- There can be more than one instance of each class held by the Event!
  - A JetChunk holds all the jets found by a single algorithm, with specific parameters.
  - Several K_T algorithms with different y_{cut} values yield several JetChunk instances.
- We need to identify specific instances, not just the class the user wants... leading to Selectors.
  - Selectors inherit from either AbsSelector or AbsTSelector<T>.
    - AbsSelector if they only need to use the AbsChunk interface: implement *bool match(const AbsChunk& chunk)*.
    - AbsTSelector<T> if they need to use functions specific to the class T: implement *bool match(const T& chunk).*

Buzzword Alert!

const type&: Used as the argument to a function, this is both an efficiency optimization (passing only an address, rather than a potentially large object) and a safety feature (the function receiving this argument cannot modify the object it is passed).

Widespread use of const

Const functions and variables are important in the DØ EDM because they allow us to have the compiler enforce the rule that a chunk, once entered into the Event, is no longer modifiable.

A dangerous scenario would be:

- tracks are found
- electrons refer to these tracks
- tracks are modified by refitting
- electrons now point to the wrong thing!

We prevent this by preventing step 3.

Memory Management

- The Event owns all the chunks it contains.
  - When the Event is deleted, all of its chunks must be deleted.
- The safe and easy way is to have the Event contain chunks by value.
  - Leads to unreasonable coupling (too much recompiling of the whole system).
  - Inefficient, because all data must be copied from where it is made into the Event.
  - Doesn't work for heterogeneous collections.

Buzzword Alert!

delete: C++ provides dynamic memory allocation, but it it must be managed by the programmer (unlike Java and Smalltalk, which are “garbage collected”). When a dynamically allocated object is no longer wanted, it is destroyed (and the memory and other resource it uses are reclaimed) by deleting the object.

Failure to delete at the right time leads to memory leaks, and the crashing of programs. Deleting at the wrong time, or more than once, leads to memory corruption, and the crashing of programs.

One must be careful with memory management.

Doing it the Hard Way

The Event has to manage pointers to AbsChunks.

- All chunk designers must be very careful to ensure that the objects they make with dynamically allocated memory have ownership given to the Event at exactly the right time -- not before (would lead to double deletions) or too late (would lead to memory leaks).
- A standard technique is used to handle the process, but only code reviews can assure the standard technique is used. The compiler doesn't help here.

A leak detecting tool (*e.g.* Purify) is essential!

Program = Many Modules

- A reconstruction (or trigger or analysis) program is built from many independent modules.
  - ⌘ Necessary for maintainability: break the problem into pieces of manageable size.
- It is important to minimize physical coupling between these modules.
  - ⌘ Test modules in isolation.
  - ⌘ Release libraries of manageable size rather than monolithic body of code; shorten the release cycle.
  - ⌘ Ability to make system out of interchangeable components.

Buzzword Alert!

physical coupling: Physical coupling is the dependence of one component or package upon another, at either compile time or link time.

component: A single header (*.h, *.hh or *.hpp) + source file (*.cc or *.cpp) pair; often contains a single class.

package: A group of related components. In the DØ software build system, one package generally creates one library. The entire DØ library currently consists of >200 packages; new ones are being born at a rate of ~20 per month.

Reco Process = Many Objects

A reconstruction (or trigger or analysis) program contains many “workers” (also called packages in the DØ framework jargon).

- Each worker does one task: finding tracks in the silicon, finding electromagnetic cluster in the calorimeter, identifying primary vertices, etc.

We want to minimize the physical coupling between these objects.

- They communicate with each other only through the Event.

Identifying What You Want

- Framework package that does reconstruction = reconstructor. Reconstructors have to specify the input they want without knowing about the class that creates that input.
  - EMID module requires a TrackChunk and a CalClusterChunk, but doesn't know about the classes that create them.
- Reconstructors specify what they want by type (*e.g.* TKey<TrackChunk>) and by specifying what algorithm and parameters they want
  - Parameters specified in compact form via RCPID.

Run Control Parameters

Each reconstructor may contain many parameters to be set at run-time:

- minimum chisquared for track fit, ...
- cone radius for jet finder, ...

These parameters are supplied at run time by an RCP object.

- RCP = collection of name/value pairs.
- Each RCP object is recorded in a database, which gives it a unique identifier (RCPID) based on the name/value pairs it contains.

RCPIDs

- A reconstructor labels the chunks it makes with the RCPID that is assigned to the set of parameters used to configure that reconstructor.
- Users can query the system at run time to get the parameters associated with any RCPID.
- Selectors can use RCPIDs to identify the chunks they are to match.
- Selectors can also use anything else the chunk designer decides is relevant.

Reconstructor Task Summary

- Get data from Event
  - ⌘Extract chunks using TKey<T> and appropriate selectors, often guided by RCPIDs.
- Do reconstruction work (using algorithm objects).
  - ⌘Create hits, tracks, muon candidates, or whatever is the task at hand.
- Create output chunk(s)
  - ⌘Put the created objects into the appropriate chunk, and label it with your RCPID.
- Insert the chunk(s) into the Event

Save the State

- The ability to save the state of reconstruction at any point can be very important
  - ⌘Useful in verifying the correctness of the reconstruction system.
  - ⌘Can assemble larger jobs out of smaller pieces.
- Since reconstructors communicate only through the Event, saving the state at any time is made simple.

Bookkeeping

In order to trace the history of reconstruction

- Each chunk contains the appropriate RCPIDs.
- Each chunk is issued a ChunkID, unique within that Event -- assigned when the chunk is inserted into the Event.
- Each chunk contains the ChunkIDs of its “parents”.

Since no chunk can be altered after it is inserted into the Event, the genesis of each chunk is preserved automatically.

Bookkeeping & Flexibility

Because we keep track of the genesis of each chunk, we can have more than one instance of each class.

- Tracks from **trf++** with parameter set A.
- Tracks from **trf++** with parameter set B.
- Calorimeter cell transverse energies calculated with respect to vertex found by Algorithm A.
- Calorimeter cell transverse energies calculated with respect to vertex found by Algorithm B.

This flexibility is especially useful while studying new algorithms, and comparing to old.

Work Smoothly With Others

The EDM has to work smoothly with at least two other systems:

- ⌘Event processing framework.
- ⌘Persistency mechanism.

The DØ framework was written to know nothing about the Event; the Event knows nothing of the framework. They cannot interfere.

The persistency mechanism was designed to know nothing of the Event, so the Event has to know about persistency.

DØOM

The DØ persistency mechanism is called DØOM (the DØ object model).

It provides a preprocessor that generates the code required for reading and writing classes.

All classes that are capable of persistence have to live within its guidelines.

- These requirements are placed on the implementation of the classes, not just on their interfaces.
- The Event and all chunks must obey DØOM's rules.

The Dirty Laundry

What are the weakest points? (In my opinion)

- Some rules are complex, and not enforced by the compiler. I'd prefer to have the compiler enforce all the rules.
- We make heavy use of templates. While this can give simultaneous efficiency and (type) safety, it makes for pain in the software building system.
- The EDM and DØOM were developed with too little collaboration. I'd prefer to see the persistency mechanism make use of the interfaces of the persistent classes, rather than relying on their implementations.